I. INTRO TO C

Hello World

```c
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

- **#include**: preprocessor inserts stdio.h contents
- stdio.h: contains printf declaration
- main: program starts here
- **void**: keyword for argument absence
- { }: basic block/scope delimiters
- printf: prints to the terminal
- \n: newline character
- **return**: leave function, return value

Compiling

```
$ gcc hello.c –o hello
$ ./hello
Hello World!
```

Basic Data Types

- **char** c = 5; **char** c = 'a';
  one byte, usually for characters (1970: ASCII is fine)
- **int** i = 5; **int** i = 0xf; **int** i = 'a';
  usually 4 bytes, holds integers
- **float** f = 5; **float** f = 5.5;
  4 bytes floating point number
- **double** d = 5.19562
  8 bytes double precision floating point number

Basic Data Types − logic

- **int** i = 5 / 2; //i = 2
  integer logic, no rounding
- **float** f = 5.0f / 2; //f = 2.5f
  decimal logic for float and double
- **char** a = 'a'/ 2 //a = 97 / 2 = 48
  char interpreted as character by console

Basic Data Types − signed/unsigned

- **signed int** i = –5 //i = –5 (two's complement)
- **unsigned int** i = –5 //i = 4294967291

Basic Data Types − short/long

- **short int** i = 1024 //–32768...32767
- **long int** i = 1024 //–2147483648...2147483647

Basic Data Types − more size stuff

- **sizeof int**; **sizeof long int**; //4; 4; (x86 32-Bit)
- use data types from inttypes.h to be sure about sizes:

  ```c
  #include <inttypes.h>
  int8_t i; uint32_t j;
  ```

Basic Data Types − const/volatile

- **const int** c = 5;
  i is constant, changing it will raise compiler error
- **volatile int** i = 5;
  i is volatile, may be modified elsewhere (by different program in shared memory, important for CPU caches, register, assumptions thereof)

Variables − local vs. global

```c
int m; // global variable

int myroutine(int j) {
  int i = 5 // local variable
  i = i+j;
  return i;
}
```

- *global variables* (**int** m):
  lifetime: while program runs
  placed on pre-defined place in memory
- basic block/function-local variables (**int** i):
  lifetime: during invocation of routine
  placed on stack or in registers

Variables − local vs. static

```c
int myroutine(int j) {
  static int i = 5;
  i = i+j;
  return i;
}

k = myroutine(1); // k = 6
k = myroutine(1); // k = 7
```

- static function-local variables:
  saved like global variables
  variable persistent across invocations
  lifetime: like global variables

Printing

```c
int i = 5; float f = 2.5;
printf("The numbers are i=%d, f=%f", i, f);
```

- comprised of format string and arguments
- may contain format identifiers (%d)
- see also man printf
- special characters: encoded via leading backslash:
  \n newline
  \t tab
  \' single quote
  \" double quote
  \0 null, end of string

Compound data types

- **struct**ure: collection of named variables (different types)
- **union**: *single* variable that can have multiple types
- members accessed via . operator

```c
struct coordinate {
  int x;
  int y;
}

union longorfloat {
  long l;
  float f;
}

struct coordinate c;
c.x = 5;
c.y = 6;

union longorfloat lf;
lf.l = 5;
lf.f = 6.192;
```

Functions

encapsulate functionality (*reuse*)

code structuring (*reduce complexity*)

must be **declared** and **defined**

Declaration: states signature

Definition: states implementation (implicitly declares function)

```c
int sum(int a, int b); // declaration

int sum(int a, int b) { // definition
  return a+b;
}
```

Header files

header file for frequently used declarations

use **extern** to declare global variables defined elsewhere

use **static** to limit scope to current file (e.g. **static float** pi in sum.c: no pi in main.c)

```c
// mymath.h
int sum(int a, int b);
extern float pi;

// sum.c
#include "mymath.h"

float pi = 3.1415927;
int sum(int a, int b) {
  return a+b;
}

// main.c
#include <stdio.h>
#include "mymath.h"

void main() {
  printf("%d\n", sum(1,2));
  printf("%f\n", pi);
}
```

Data Segments and Variables

Stack: local variables

Heap: variables crated at runtime via malloc()/free()

Data Segment: static/global variables

Code: functions

Function overloading

no function overloading in C!

use arrays ore pointers

Pointers

```c
int a = 5;
int *p = &a // points to int, initalized to point to a
int *q = 32 // points to int at address 32
int b = a+1;
int c = *p; // dereference(p) = dereference(&a) = 5
int d = (*p)+2 // = 7
int *r = p+1; // pointing to next element p is pointing to
int e = *(p+2) // dereference (p+2) = d = 7
```

Pointers − linked list

linked-list implementation via next-pointer

```c
struct ll {
  int item;
  struct ll *next;
}

struct l first;
first.item = 123;

struct ll second;
second.item = 456;
first.next = &second;
```

Arrays

= fixed number of variables *continuously laid out in memory*

```c
int A[5]; // declare array (reserve memory space)
A[4] = 25; A[0] = 24; // assign 25 to last, 24 to first elem
char c[] = {'a',5,6,7,'B'} // init array, length implicit
c[64] = 'Z' // NO bounds checking at compile/run (may raise
    protection fault)

// declare pointer to array; address elements via pointer:
char *p = c;
*(p+1) = 'Z'; p[3] = 'B'; char b = *p; // = 'a'
```

Strings

= array of **char**s terminated by NULL:

```c
char A[] = { 'T', 'e', 's', 't', '\0' };
char A[] = "Test";
```

declaration via pointer:
```c
const char *p = "Test";
```

common string functions (string.h):

length: size_t strnlen(const char *s, size_t maxlen)

compare:
  int strncmp(const char *s1, const char *s2, size_t n);

copy: int strncpy(char *dest, const char *src size_t n);

tokenize: char *strtok(char *str, const char *delim);
    (e.g. split line into words)

Arithmetic/bitwise operators

arithmetic operators:
  a+b, a++, ++a, a+=b, a−b, a−−, −−a, a−=b, a∗b, a∗=b, a/b, a/=b, a%b, a%=b

logical operators:
  a&b, a|b, a>>b, a<<b, a^b, ~a

difference pre-/post-increment:

```c
int a = 5;
if(a++ == 5) printf("Yes"); // Yes
a = 5;
if(++a == 5) printf("Yes"); // nothing
```

operators in order of precedence:
```
(), [], ->, .
!, ++, −−, +y, −y, *z, &=, (type), sizeof
*, /, %
+, −
<<, >>
<, <=, >, >=
==, !=
&
^
|
&&
||
?, :
=, +=, −=, *=, /=, %=, &=, ~=,=, «=, »=|
,
```

Structures

brackets only needed for multiple statements

if/else, **for**, **while**, **do−while**, **switch**

may use **break**/**continue**

switch: need break statement, otherwise will fall through

```c
if(a==b) printf("Equal") else printf("Different");
for(i=10; i>=10; i−−) printf("%d", i+1);
int i=10; while(i−−) printf("foo");
int i=0; do printf("bar"); while(i++ != 0);

char a = read();
switch(a) {
  case '1':
    handle_1();
    break;
  default:
    handle_other();
    break;
}
```

Type casting

explicit casting: precision loss possible
```c
int i = 5; float f = (float)i;
```

implicit casting: if no precision is lost
```c
char c = 5; int i = c;
```

pointer casting: changes address calculation
```c
int i = 5; char *p = (char *)&i; *(p+1)= 5;
```

type hierarchy: „wider"/„shorter" types
```c
unsigned int wider than signed int
```
operators cast parameters to widest type
Attention: assignment cast after operator cast

C Preprocessor

modifies *source code* before compilation

based on preprocessor *directives* (usually starting with `#`)

`#include <stdio.h>`, `#include "mystdio.h"`:
copies contents of file to current file

only works with strings in source file

completely ignores C semantics

Preprocessor − search paths

`#include <file>`: system include, searches in:
```
/usr/local/include
libdir/gcc/[target]/[version]/include
/usr/[target]/include
/usr/include
```
(target: arch-specific (e.g. i686-linux-gnu),
version: gcc version (e.g. 4.2.4))

`#include "file"`: local include, searches in:
directory containing current file
then paths specified by `-i <dir>`
then in system include paths

Preprocessor − definitions

defines introduce replacement strings (can have arguments, based on string replacement)

can help code structuring, often leading to source code cluttering

```c
#define PI 3.14159265
#define TRUE (1)
#define max(a,b) ((a > b) ? (a) (b))
#define panic(str) do { printf(str); for (;;) } while(0);

#ifdef __unix__
# include <unistd.h>
#elif defined _WIN32
# include <windows.h>
#endif
```

Preprocessor − predefined macros

system-specific:
`__unix__, _WIN32, __STDC_VERSION__`

useful:
`__LINE__, __FILE__, __DATE__`

Libraries

= collection of functions contained in object files, glued together in dynamic/static library

ex.: Math header contains declarations, but not all definitions
⤳ need to link math library: `gcc math.c -o math -lm`

```c
#include <math.h>
#include <stdio.h>

int main() {
  float f = 0.555f;
  printf("%f", sqrt(f*4));
  return 0;
}
```

II.   INTRODUCTION TO OPERATING SYSTEMS

What's an OS?

abstraction: provides abstraction for applications
manages and hides hardware details
uses low-level interfaces (not available to applications)
multiplexes hardware to multiple programs (*virtualisation*)
makes hardware use efficient for applications

protection:
from processes using up all resources (*accounting*, *allocation*)
from processes writing into other processes memory

resource managing:
manages + multiplexes hardware resources
decides between conflicting requests for resource use
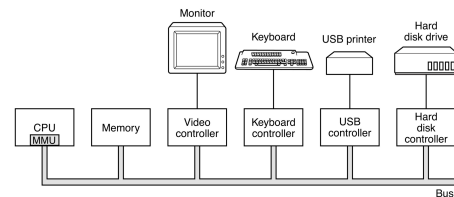strives for efficient + fair resource use

control:
controls program execution
prevents errors and improper computer use
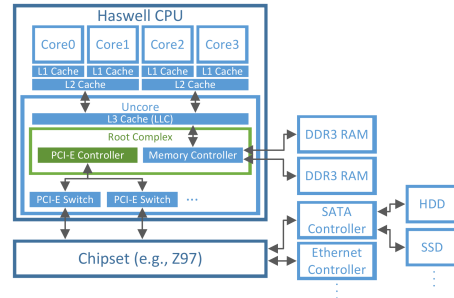
⤳ **no universally accepted definition**

Hardware Overview

CPU(s)/devices/memory (conceptually) connected to common bus

CPU(s)/devices competing for memory cycles/bus

all entities run concurrently

today: multiple busses

Central Processing Unit (CPU) − Operation

fetches instructions from memory, executes them (instruction format/-set depends on CPU)

CPU internal registers store (meta-)data during execution (general purpose registers, floating point registers, instruction pointer (IP), stack pointer (SP), program status word (PSW),...)
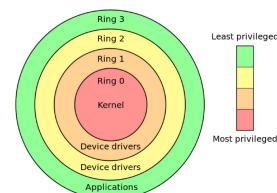
execution modes:
user mode (x86: *Ring 3/CPL 3*):
only non-privileged instructions may be executed
cannot manage hardware → **protection**
kernel mode (x86: *Ring 0/CPL 0*):
all instructions allowed
can manage hw with **privileged instructions**

Random Access Memory (RAM)

keeps currently executed instructions + data

today: CPUs have built-in *memory controller*

root complex connected directly via
„wire" to caches
pins to RAM
pins to PCI-E switches

Caching

RAM delivers instructions/data slower than CPU can execute

memory references typicalle follow *locality principle*:
spatial locality: future refs often near previous accesses
(e.g. next byte in array)
temporal locality: future refs often at previously accessed ref
(e.g. loop counter)

caching helps mitigating this memory wall:
copy used information temporarily from slower to faster storage
check faster storage first before going down **memory hierarchy**
if not, data is copied to cache and used from there

Access latency:
register: ∼1 CPU cycle
L1 cache (per core): ∼4 CPU cycles
L2 cache (per core pair): ∼12 CPU cycles
L3 cache/LLC (per uncore): ∼28 CPU cycles (∼25 GiB/s)
DDR3-12800U RAM: ∼28 CPU cycles + ∼ 50ns (∼12 GiB/s)

Caching − Cache Organisation

caches managed in hardware

divided into *cache lines* (usually 64 bytes each, unit at which data
is exchanged between hierarchy levels)

often separation of data/instructions in faster caches (e.g. L1, see
harward architecture)

cache hit: accessed data already in cache (e.g. L2 cache hit)

cache miss: accessed data has to be fetched from lower level

cache miss types:
compulsory miss: first ref miss, data never been accessed
capacity miss: cache not large enough for process working set
conflict miss: cache has still space, but collisions due to
placement strategy

Interplay of CPU and Devices

I/O devices and CPU execute concurrently

Each device controller
- is in charge of particular device
- has local buffer

Workflow:

1. CPU issues commands, moves data to devices
2. Device controller informs APIC that operation is finished
3. APIC signals CPU
4. CPU receives device/interrupt number from APIC, executes
handler
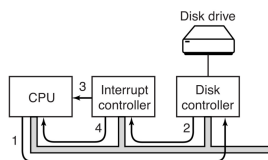
Device control

Devices controlled through their **device controller**, accepts com-
mands from OS via **device driver**

devices controlled through device registers and device memory:
control device by writing device registers
read status of device by reading device registers
pass data to device by reading/writing device memory

2 ways to access device registers/memory:

1. **port-mapped IO** (PMIO):
use special CPU instructions to access port-mapped
registers/memory
e.g. x86 has different in/out-commands that transfer
1,2 or 4 bytes between CPU and device
2. **memory-mapped IO** (MMIO):
use same address space for RAM and device memory
some addresses map to RAM, others to different devices
access device's memory region to access device registers/-
memory

some devices use hybrid approaches using both

Device control − Nvidia general purpose GPU

memory-mapped ring-buffer and put/get-device

mapping can be exposed to application ⇝ application can submit
commands in user-mode
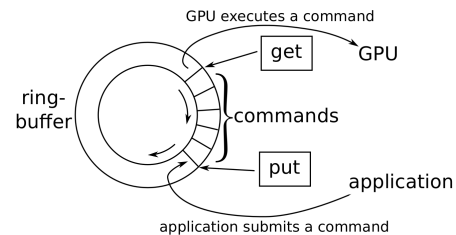
Summary

The OS is an abstraction layer between applications and
hardware (multiplexes hardware, hides hardware de-
tails, provides protextion between processes/users)

The CPU provides a separation of User and Kernel mode
(which are required for an OS to provide protection
between applications)

CPU can execute commands faster than memore can de-
liver instructions/data − memory hierarchy mitigates
this memory wall, needs to be carefully managed by
OS to minimize slowdowns

device drivers control hardware devices through
PMIO/MMIO

Devices can signal the CPU (and through the CPU notify
the OS) through interrupts

III. OS CONCEPTS

OS Invokation

OS Kernel does **not** always run in background!

Occasions invoking kernel, switching to kernel mode:

1. **System calls**: User-Mode processes require higher privileges
2. **Interrupts**: CPU-external device sends signal
3. **Exceptions**: CPU signals unexpected condition

System calls − motivation

Problem: protect processes from one another

Idea: Restrict processes by running them in user-mode

⇝ Problem: now processes cannot manage hardware,...
who can switch between processes?
who decides if process may open certain file?

⇝ Idea: OS provides **services** to apps
app calls system if service is needed (**syscall**)
OS checks if app is allowed to perform action
if app may perform action and hasn't exceeded quota,
OS performs action in behalf of app in kernel mode

System Calls − Examples

fd = open(file, how,...) − open file for read/write/both

documented e.g. in man 2 write

overview in man 2 syscalls

System Calls vs. APIs

syscalls: interface between apps and OS services, limited number
of well-defined entry points to kernel

APIs: often used by programmers to make syscalls
e.g. printf library call uses write syscall

common APIs: Win32, POSIX, C API

System Calls − implementation

trap instruction: single syscall interface (entry point) to kernel
switches CPU to kernel mode, enters kernel in same, predefined
way for all syscalls
system call dispatches then acts as syscall multiplexer

syscalls identified by number passed to trap instruction
syscall table maps syscall numers to kernel functions
dispatcher decides where to jump based on number and table
programs (e.g. stdlib) have syscall number compiled in!
⇝ never reuse old numbers in future kernel versions

Interrupts

devices use interrupts to signal predefined conditions to OS
reminder: device has „interrupt line" to CPU
e.g. device controller informs CPU that operation is finished

programmable interrupt controller manages interrupts
interrupts can be **masked**
masked interrupts: queued, delivered when interrupt unmasked
queue has finite length ⇝ interrupts can get lost

noteable interrupt examples:

1. *timer-interrupt*: periodically interrupts processes, switches
to kernel ⇝ can then switch to different processes for fairness
2. *network interface card* interrupts CPU when packet was
received ⇝ can deliver packet to process and free NIC
buffer

when interrupted, CPU

1. looks up **interrupt vector** (= table pinned in memory,
contains addresses of all service routines)
2. transfers control to respective **interrupt service routine**
in OS that handles interrupt

interrupt service routine must first save interrupted processe's
state (instruction pointer, stack pointer, status word)

Exceptions

sometimes unusual condition makes it impossible for CPU to con-
tinue processing

⇝ **Exception** generated within CPU:

1. CPU interrupts program, gives kernel control
2. kernel determines reason for exception
3. if kernel can resolve problem ⇝ does so, continues **faulting
instruction**
4. kills process if not

Difference to Interrupts: interrupts can happen in any context,
exceptions always occur asynchronous and in process context

OS Concepts − Physical Memory

up to early 60s:
- programs loaded and run directly in *physical memory*
- program too large → partitioned manually into *overlays*
- OS then swaps overlays between disk and memory
- different jobs could obeserve/modify eachother

OS Concepts − Address Spaces

bad programs/people need to be isolated

Idea: give every job the illusion of having all memory to itself
every job has own *address space*, can't name addresses of others
jobs always and only use virtual addresses

Virtual Memory − indirect addressing

Today: every CPU has built-in **memory management unit**
(*MMU*)

MMU translates virtual addresses to physical addresses at every
store/load operation

⇝ address translation protects one program from another

Definitions:
Virtual address: address in process' address space
Physical address: address of real memory

Virtual Memory − memory protection

MMU allows kernel-only virtual addresses
- kernel typically part of all address spaces
- ensures that apps can't touch kernel memory

MMU can enforce *read-only* virtual addresses
- allows safe sharing of memory between apps

MMU can enforce execute disable
- makes code injection attacks harder

Virtual Memory − page faults

not all addresses need to be mapped at all times
- MMU issues *page fault* exception when accessed virtual
address isn't mapped
- OS handles page faults by loading faulting addresses and then
continuing the program
- ⇝ memory can be **over-committed**: more memory than
physically available can be allocated to application

page faults also issued by MMU on illegal memory accesses

OS Concepts − Processes

= program in execution („instance" of program)

each process is associated with a **process control block** (*PCB*)
contains information about allocated resources

each process is associated with a virtual **address space** (*AS*)
- all (virtual) memory locations a program can name
- starts at 0 and runs up to a maximum
- address 123 in AS1 generally ≠ address 123 in AS2
- indirect addressing ⇝ different ASes to different programs
- ⇝ protection between processes

OS Concepts – address space layout

address spaces typically laid-out in different **sections**
 - memory addresses between sections **illegal**
 - illegal addresses ⤳ page fault
 - more specifically calles **segmentation fault**
 - OS usually kills process causing segmentation fault

**Stack**: function history, local variables

Data: Constants, static/global variables, strings

Text: Program code

Address (hex)
FFFF

| Stack |
| Gap |
| Data |
| Text |

0000
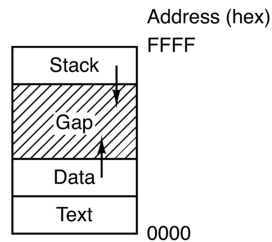
OS Concepts – Threads

each progress: ≥ 1 threads (representing execution states)

IP stores currently executed instruction (address in text section)

SP register stores address of stack top
 (> 1 threads → multiple stacks!)

PSW contains flags about execution history
 (e.g. last calculation was 0 → used in following jump instruction)

more general purpose registers, floating point registers,...

Threads

User-Space
Kernel-Space

PCB
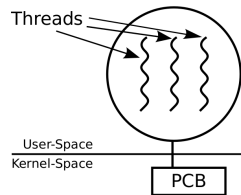
OS Concepts – Policies vs. Mechanisms

separation useful when designing OS

Mechanism: implementation of what is done
 (e.g. commands to put a HDD into standby mode)

Policy: rules which decide when what is done and how much
 (e.g. how often, how many resources are used,...)

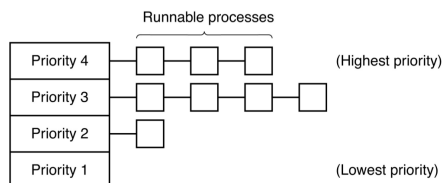→ *mechanismes can be reused even when policy changes*

OS Concepts – Scheduling

multiple processes/threads available ⤳ OS needs to switch between them (for multitasking)

scheduler decides which job to run next (policy)

dispatcher performs task-switching (mechanism)

schedulers try to
 - provide fairness
 - while meeting goals
 - and adhering to priorities

Runnable processes

Priority 4				(Highest priority)
Priority 3				
Priority 2		
Priority 1		(Lowest priority)

OS Concepts – Files

OS hides peculiarities of disks,...

programmer uses device-independent *files/directories* for persistent storage

Files: associate *file name* and *offset* with bytes

Directories: associate *directory names* with directory names or file names

File System: ordered block collection
 - main task: translate (dir name + file name + offset) to block
 - programmer uses file system operations to operate on files
   (open, read, seek)

processes can communicate directly through special *named pipe* file (used with same operations as any other file)

OS Concepts – Directory Tree

directories form *directory tree/file hierarchy*
 → structure data

root directory: topmost directory in tree

files specified by providing *path name* to file

OS Concepts – Mounting

*nix: common to orchestrate multiple file systems in single file hierarchy

file systems can be *mounted* on directory

Win: manage multiple directory hierarchies with drive letters
 (e.g. C:\Users)

OS Concepts – Storage Management

OS provides uniform view of information storage to file systems
 - *drivers* hide specific hardware devices
   → hides device peculiarities
 - general interface abstracts physical properties to logical units
   → block

OS increases I/O performance:
 - **Buffering**: Store data temporarily while transferred
 - **Caching**: Store data parts in faster storage
 - **Spooling**: Overlap one job's output with other job's input